UNCORRECTED PROOF

CHAPTER 19

Control intuition

19.1 Object picking 366
19.2 Intuitive mapping 372
19.3 Display-relative mapping 377
19.4 World-relative mapping 381
19.5 Summary 391

As you saw in the previous chapter, trying to control a target object with static control mapping can produce reasonable results, but only if you, the designer, carefully control the position of the user's view in the world. For some applications this might be acceptable, but for many it is not. When static control mapping is combined with general forms of navigation, situations often arise where the controls seem wrong or backward. Another practical aspect of trying to control objects in the scene is designating which object is the target. In the previous examples this was handled by assigning a different mouse button, or button and modifier key combination, to each object in the scene. Such an approach obviously won't work for more than a few objects, and even then it is impractical for the user to have to remember which button-modifier goes with which object. Object picking is the technique needed to handle this problem.

The previous chapters described how the framework handles the fundamental aspects of the control chain, including device input and enabling, input filtering and mapping, and target actuation. This chapter will explore those aspects of the control chain needed to achieve more intuitive control of the situation, including control enabling based on picking, and dynamic forms of coordinate mapping, specifically DRM and WRM.

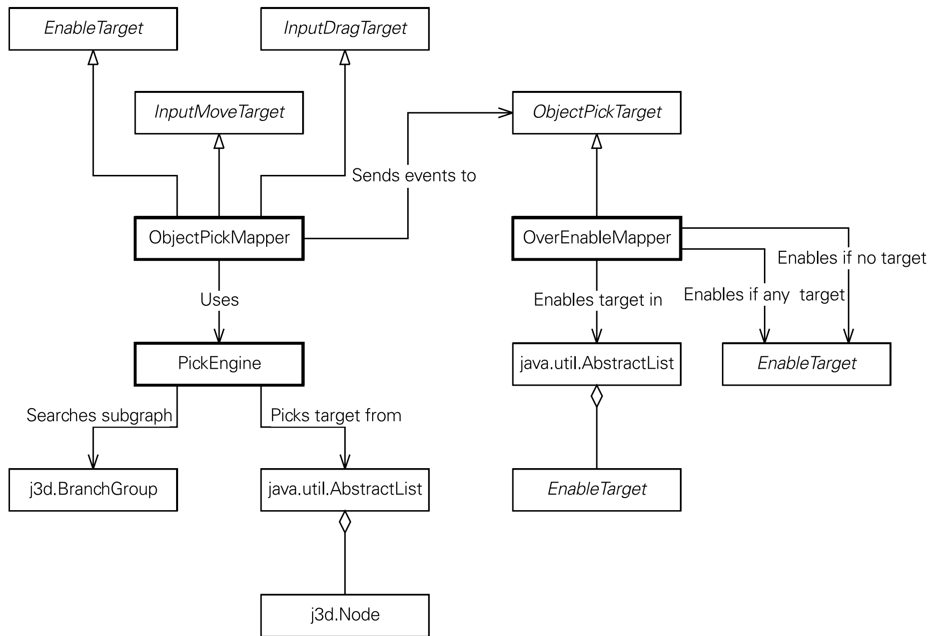

Figure 19.1 Relationship of classes and interfaces related to object picking and enabling

19.1 OBJECT PICKING

Object picking is the workhorse of a 3D UI. For the most part, nothing happens of an intuitive nature in a 3D scene without object picking. Although the framework supports picking in its various forms—discrete and continuous, bounds and geometric—the emphasis is on continuous geometric picking. Picking can be employed in a number of ways in a UI. This section introduces a general-purpose picking engine, and then describes how it can be used for target control enabling. Later chapters will describe how to use picking for other UI needs, such as for feedback and selection.

With a few minor exceptions, the classes used in picking and picking-based control enabling are found in the package `j3dui.control.mappers`. Table 19.1 summarizes the classes and event interfaces related to picking and enabling, and figure 19.1 illustrates how they cooperate.

Table 19.1 Classes and interfaces related to object picking and enabling

Class/*Interface*	Description	Input Events	Output Events
ObjectPickTarget	Event interface for reporting pick target changes		
ObjectPickSplitter	Object pick event splitter	ObjectPickTarget	ObjectPickTarget
PickEngine	Consolidates core picking functionality into a single class. Based on target list picking.		

Table 19.1 Classes and interfaces related to object picking and enabling (continued)

Class/*Interface*	Description	Input Events	Output Events
ObjectPickMapper	Maps input movement and drags into object picks using a pick engine	EnableTarget InputMoveTarget InputDragTarget	ObjectPickTarget
OverEnableMapper	Maps object picks into target enables	ObjectPickTarget	EnableTarget (list) EnableTarget (any) EnableTarget (none)

19.1.1 PickEngine class

The complete Java 3D picking model was described in detail in chapter 13. As you may recall, the Java 3D picking process is complex and decentralized. To achieve a degree of flexibility and generality in what can be a target object for picking, the framework uses an approach where the candidate pick objects are specified explicitly in a target list. Only objects, or more specifically scene graph Node objects, that are in the list can be picked, and then only if the objects and their descendants have been properly configured, with which the framework also assists.

The heart of the framework's picking capability is the PickEngine class. The constructor requires a root BranchGroup object in the scene graph under which the pick search will be performed, and a list of pick target objects that will be sought. A pick operation is performed when the method `pickTarget` is called, which requires specification of the source display and the pick cursor position in that display. The method returns the index of the target in the target list that was hit, if any, and optionally the 3D hit position on the target object. The method `getPickTarget` can be used to convert the pick index result into its equivalent pick object reference. After construction, the pick root and target list can be changed with the `setRoot` and `setTargets` methods, respectively.

Target pickability

For a target node to be pickable, it must lie under the pick root in the scene graph, its pick reporting capability must be set, its leaf shape nodes must be set pickable, and the capability to intersect the leaf shape geometry must be set. By default, Java 3D enables leaf nodes for picking, but pick reporting and geometry intersection must be explicitly enabled. As a convenience, objects in the target list are automatically configured for pick reporting, but not for leaf node picking. For explicit control of leaf node picking, you can use the utilities in the PickUtils class found in the `j3dui.utils` package. Also, many of the framework shape objects include the ability to set full leaf pickablity.

Engine configuration

By default, a pick engine performs geometry picking, and quits searching after the first pickable object is hit, whether or not it is in the target list. Bounds-only picking can be

specified with the `setUseBounds` method, and hit testing of all objects under the pick cursor can be requested with the `setHitAll` method. Because picking performance is crucial to good UI design, the pick engine also includes provisions for assessing picking performance. A cheat mode, enabled with the `setUseCheat` method, allows side-by-side comparison of in-scene picking performance with and without Java 3D picking. This way, any difference in performance is due to the picking process itself, not the application overhead of handling input events and actuating targets.

Target order

The order of targets in the target list can be significant. If a parent and a descendant object, such as a desk and a drawer in the desk, are both targets in the target list, pick reporting will be enabled for both nodes. If a pick occurs on the descendant, both the parent and the descendant will be included in the hit's scene graph path. For each hit, the pick engine searches the scene graph path for each target in the target list, starting with the first target, which defines the order of pick precedence. When a hit on a list target is found, the search quits. Thus, for the descendant (the drawer) to be picked instead of its parent (the desk) in this situation, the descendant must be in the target list before the parent.

19.1.2 ObjectPickMapper class

The pick engine is truly only an engine; it does not handle or generate any events. Event generation is, instead, the duty of the ObjectPickMapper class, which converts input movement and drags into target object picks, including "no target" picks. The picking itself is handled through delegation to a PickEngine object configured with the desired pick root and target list. Both continuous and discrete picking modes are supported. For continuous picking, pick events are reported only when the target object changes, and not when the pick cursor moves, which helps to keep event handling overhead to a minimum.

Picking modes

To accommodate various flavors of discrete and continuous picking, several methods are provided. These methods affect only the mode of pick reporting for input drag events. Continuous pick reporting, as a result of input movement events, is not affected by these modes. The different modes can be combined, with the result being an ORing of object pick events. The `setDoPick` method specifies that the pick target is reported when a drag starts, and no target is reported when the drag stops. This is the default mode. The `setDoDrag` method specifies that the pick target is reported only when the drag starts, which is useful for triggering the drag portion of a drag-and-drop operation. The `setDoDrop` method specifies that the pick target is reported continuously during the drag, and when the drag stops, which is useful for tracking drop targets.

ObjectPickTarget interface

Pick reporting requires a new event interface. The `ObjectPickTarget` interface reports both the pick index and the pick object. In general, the pick index should be used if the event target has knowledge of the pick engine's target list, because it uniquely identifies the object in the list (and the object reference may not). If the event target is not privy to the target list, or if the pick event originates from a pick process that does not use a list, then the pick object must be used. It is important to keep these alternative uses of the event in mind when generating or handling `ObjectPickTarget` events because some event targets use one or the other exclusively. As with all other framework events, the event has a corresponding ObjectPickSplitter class.

19.1.3 OverEnableMapper class

Knowing which object was picked using a pick engine and an object pick mapper is only two thirds of the solution when it comes to mouseover object control. The framework provides the last third of the solution in the form of the OverEnableMapper class. To use this class, you have to create a list of control targets that implement the `EnableTarget` interface, with a one-to-one correspondence between the event targets in this list and the pick targets in the pick engine's target list, such as that used by an ObjectPickMapper object. When a new target is picked, as indicated by the input `ObjectPickTarget` event interface, any previous enable target is disabled and the one corresponding to the new pick is enabled. Note that OverEnableMapper uses only the index parameter in its input event interface.

For convenience, two special-purpose control targets can be set in addition to, or in lieu of, the event list targets. The `setEventTargetAny` method establishes an enable target that is notified when any target is picked; the `setEventTargetNone` method establishes an event target that is notified when no target is picked.

19.1.4 Example: OverEnabling

This example demonstrates object picking and mouseover enabling of control inputs.

See

The virtual world contains four target objects: three in a central column (red, green, blue) and one to the right (magenta). A screen shot is provided in figure 19.2.

Do

- Drag the mouse (left button with SHIFT) in the display or use the arrow keys (with SHIFT) to orbit the view in heading and elevation about the world origin.
- Drag the mouse (left button) on the top (red) target or use the ARROW keys to translate it along the world *X-Y* axes.

- Drag the mouse (left button) on the middle (green) target or use the ARROW keys to rotate it about the world *Y* axis.
- Drag the mouse (left button) on the bottom (blue) target or use the ARROW keys to scale it along the world *x-y* axes.
- Drag the mouse (left button) on the right (magenta) target or use the ARROW keys to roll the target like a trackball.
- Repeat the drag operations from different view directions.

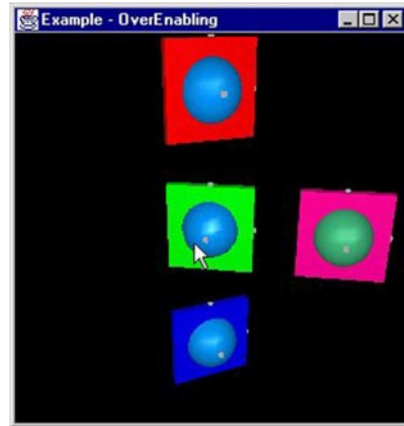

Figure 19.2 Screen shot of the OverEnabling example

Observe

- All target operations use mouseover to enable the operation.
- Target picking can differentiate between parent (green thing) and child (magenta thing) in the scene graph.
- All target operations use the first mouse button and no MODIFIER keys.
- All coordinate mapping is static, meaning that manipulation of targets will seem nonintuitive if the view is not head-on.

The general structure of the code for this example is similar to that in the previous example, which demonstrated actuator groups, but with one important difference. In this example, the utility building block buildOverEnabler from the IntuitiveBlocks class is used to perform overenabling of the target actuators. The first code sample below is from the OverEnabling example class. It shows those portions that deal with the translation target and overenabling. The second sample is from the utility class. It shows how the utility building block combines the framework's core building blocks for picking with move and drag sensors into a single convenient module for overenabling.

Translation control and overenabling

Filename: J3duiBook/examples/OverEnabling/OverEnabling.java

```
...
     // setup manipulation targets
     /// translation target
     AffineGroup affineXlt = new AffineGroup(new TestThing(
      TestThing.BOX_TRANSLATION, TestThing.BALL_DEFAULT));
     getWorld().addSceneNode(BasicBlocks.buildTarget(
      affineXlt, new Vector3d(0, 3, 0))[0]);
     ...
     // setup manipulation controls
     InputDragMapper mapper;
```

```
    /// translation, first button
    mapper = ActuationBlocks.buildDirectMapper(
     affineXlt.getTranslation(), new Vector2d(.025, .025),
     Mapper.DIM_X, Mapper.DIM_Y, true);

    ActuationBlocks.buildRelativeDragger(mapper,
     getView(), Input.BUTTON_FIRST,
     Input.MODIFIER_NONE, Input.MODIFIER_NONE);
    …
    // setup over management
    ArrayList pickList, enableList;

    pickList = new ArrayList();
    pickList.add(affineXlt);
    pickList.add(sphere);
    pickList.add(affineRot);
    pickList.add(affineScl);

    enableList = new ArrayList();
    enableList.add(affineXlt);
    enableList.add(sphere);
    enableList.add(affineRot);
    enableList.add(affineScl);

    IntuitiveBlocks.buildOverEnabler(getView(),
     getWorld().getSceneRoot(), pickList, enableList);
  …
```

Utility block combining core blocks for overenabling

Filename: J3duiBook/lib/j3dui/utils/blocks/IntuitiveBlocks.java

```
…
  /**
  Creates mouse move and drag sensors, an object picker, and
  an over enabler and connects them together for continuous
  target picking and enabling, whether during mouse movement
  or dragging.  All enable targets will be initialized as if
  no target was picked.
  @param view Source display.
  @param pickRoot Scene graph pick root.
  @param pickList List of pick targets.
  @param enableList List of enable targets corresponding to
  the targets in the pick list.
  */
  public static final OverEnableMapper buildOverEnabler(
   AppView view, BranchGroup pickRoot, ArrayList pickList,
   ArrayList enableList) {

    // setup over target enabling
    OverEnableMapper enabler = new OverEnableMapper();
    enabler.setEventTargets(enableList);
```

```
    enabler.initEventTargets(-1);

    // setup continuous target picking
    ObjectPickMapper picker = new ObjectPickMapper(enabler,
     new PickEngine(pickRoot, pickList));

    /// generate mouse position for any move or drag
    MouseDragSensor dragger = new MouseDragSensor(picker,
     view.getDisplay(), view.getRoot());
    dragger.setButtons(Input.BUTTON_ALL);

    MouseMoveSensor mouseMove = new MouseMoveSensor(
     picker, view.getDisplay(), view.getRoot());

    return enabler;
  }
…
```

19.2 INTUITIVE MAPPING

As described in chapter 4, intuitive coordinate mapping, such as DRM and WRM, utilize control source and target spaces. In many respects, the InputDragMapper class described in an earlier chapter was a shortcut that went from 2D device input straight to 4D actuation output, completely bypassing the intermediate 3D source and target spaces required for intuitive mapping. The resulting control chain got things moving, so to speak, but not in a very intuitive manner. This section takes a more in-depth look at source and target spaces, and describes the framework's support for a longer and more effective control chain, with additional links that permit more generalized and, as a result, more intuitive interpretation of control inputs.

19.2.1 Source and target spaces

The seeds for understanding the role of the source and target space in intuitive mapping were planted in both parts 2 and 3 of the book. Chapter 4 discussed the general problem, and chapter 13 addressed the specific notion of world versus local spatial coordinates, and how local-to-local transforms can be useful. The matter boils down to one of spatial interpretation. When the user drags a mouse in the display, is the drag simply a change in mouse *X* and *Y* value, or should it be interpreted as a drag across a horizontal floor or a vertical wall in the context of the scene? When the control input is applied to an actuator, is it simply applied with respect to the target's local coordinate system, or should it be applied relative to the floor that the object is sitting on, or the wall to which it is attached?

Source space

When the user drags the mouse over a floor or wall in the scene, rather than thinking of the drag as a change in mouse *X* and *Y* value, it can be interpreted relative to the 3D space defined by those objects. Depending on which object is under the mouse, the 2D drag can be interpreted as being parallel to the horizontal floor or the vertical wall, in 3D space. This 3D motion could be considered a generalized form of drag input, as if we weren't dealing with a POCS and the user had waved a 3D mouse through the scene, over the floor or along the wall. As with its 2D counterpart, this generalized 3D drag input can be filtered, such as for absolute and relative origin, clamping and scaling, and even gestures, only in 3D.

An example using third-person controls might help to further illustrate the role of the 3D source space in the control chain. Normally we think of third-person controls as being virtual knobs and sliders in the application window. The user is expected to manipulate these controls as if they were pasted to the display screen. Now imagine that these controls were instead pasted to the floor and walls in the 3D scene. Using a "3D mouse," the user could reach into the scene and turn a knob on the floor or slide a knob on the wall. Although the user interaction is occurring in 3D, the effect on the actuation targets of the controls is the same as in the 2D case. The notion of a source space is simply a more generalized way to specify what a control drag is in reference to. Instead of the source space always being the display plane, it can now be any plane in the virtual world. In fact, it doesn't even have to be a plane. It could be the rolling surface of 3D terrain or the surface of an arbitrarily complex 3D shape.

Target space

As defined in the previous chapter, the actuation space is a 4D space that directly controls the actuator target's dynamic state. In a translation actuator connected to a transform group, when the actuation *X* value changes by 10 units, the target's transform state is modified accordingly, with the target group's children moving 10 units in a direction defined by the group's local coordinate space. If you instead wanted the *X* input to move the target relative to some other space in the scene, such as the floor or a wall, then a spatial transform would be needed, with the target space defined by the local coordinate system of the target object. In the general case, a local-to-local transform would convert the input source value from its local coordinate space to that of the target's.

19.2.2 Control chain spaces

Putting the source and target pieces together, there are a total of four spaces: 2D input, 3D source, 3D target, and 4D actuation. Correspondingly, there are three coordinate mappings: The 2D device input is interpreted relative to some 3D source space, converting it into a 3D source input. In turn, this generalized 3D source value is converted into a generalized 3D target value by means of a local-to-local spatial transform, such as that defined in chapter 13 using the `getLocalToVworld`

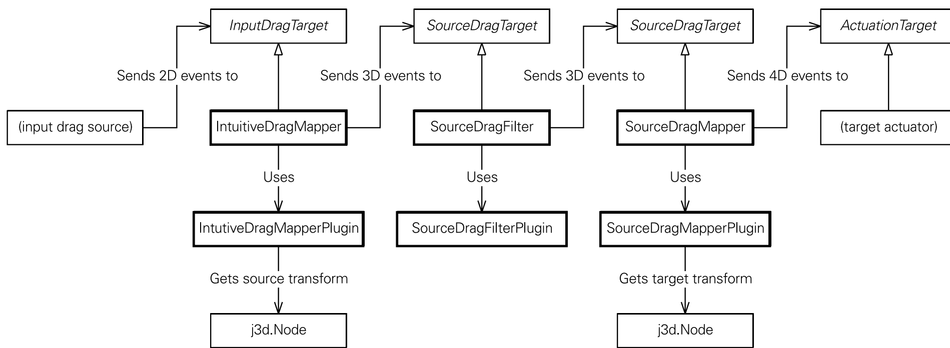

Figure 19.3 Relationship of classes and interfaces related to intuitive coordinate mapping

methods of a source and target Java 3D Node object. The target value is then mapped into an actuation value, similar to that done by InputDragMapper, only here it is from 3D to 4D space instead of from 2D to 4D space. To simplify things, the framework exposes only the source space for processing with filter blocks, and the target and actuation transformations are combined into a single building block, with only the input source and output actuation interfaces exposed.

All of the classes discussed here are located in the `j3dui.control.mappers.intuitive` package. Table 19.2 summarizes the basic classes and interfaces that the framework provides for generalized intuitive coordinate mapping. Figure 19.3 shows the relationship of the building blocks.

Table 19.2 Classes and interfaces related to intuitive coordinate mapping

Class/*Interface*	**Description**	**Input Events**	**Output Events**
IntutiveDragMapper	A coordinate mapper for input drag to source drag conversion. Uses a plug-in to perform a specific type of coordinate mapping	InputDragTarget	SourceDragTarget
IntuitiveDragMapperPlugin	The abstract base class for intuitive drag mapper plug-ins		
SourceDragTarget	Event interface for reporting source drag gestures		
SourceDragSplitter	Source drag event splitter	SourceDragTarget	SourceDragTarget
SourceDragFilter	The source drag filter, which uses a plug-in to perform a specific filtering operation	SourceDragTarget	SourceDragTarget
AbsoluteSourceDragPlugin	A source drag filter plug-in that interprets the drag with an absolute drag origin		

Table 19.2 Classes and interfaces related to intuitive coordinate mapping (continued)

Class/*Interface*	Description	Input Events	Output Events
RelativeSourceDragPlugin	A source drag filter plug-in that interprets the drag with a relative drag origin		
SourceDragMapper	A coordinate mapper for source drag to target drag to actuation conversion. Uses a plug-in to perform a specific type of coordinate mapping	SourceDragTarget	ActuationTarget
SourceDragMapperPlugin	The abstract base class for source drag mapper plug-ins		
DirectSourceDragPlugin	A source drag mapper plug-in that performs source scale and target offset, and that maps source dimensions to none, one, or more output dimensions		
AxisAngleSourceDragPlugin	A source drag mapper plug-in that maps a rotation about a fixed source axis to a rotation about the corresponding target axis		

In the following discussion of the framework's building blocks for intuitive mapping, you'll notice a number of similarities with earlier building blocks. This should come as no surprise because many of the same operations still need to be performed, such as input drag mapping, output actuation mapping, and drag filtering, only here these operations involve 3D source space instead of 2D input space. Also, source and target spaces are defined in terms of nodes living in those spaces. The building blocks automatically configure such nodes for live access of their local-to-Vworld transforms, and make extensive use of the framework's local-to-local spatial transform utilities—the `Mapper.toTargetSpace` methods—to perform source-to-target spatial transforms.

19.2.3 IntuitiveDragMapper class

Conversion from 2D input space to 3D source space is performed by the IntuitiveDragMapper class. It uses an IntuitiveDragMapperPlugin to handle the details of translating between spatial coordinate systems and event types. Subclasses of this plug-in are provided to support DRM and several flavors of WRM. Each handles acquisition and use of source space reference nodes differently. Specifics will be covered in later sections on DRM and WRM. The internal interface between the mapper and its plug-in are similar to those between the InputDragMapper class and its plug-in, the InputDragMapperPlugin.

By definition, input to source mapping is a point mapping, meaning that the input position is mapped to a source position using a point transformation rather than

a vector transformation. The distinction is subtle, but important when setting up an input drag filter chain to feed an intuitive mapper. The mapper's input event interface is `InputDragTarget`, and its output event interface is `SourceDragTarget`.

SourceDragTarget interface

The `SourceDragTarget` interface is essentially a 3D version of the 2D `InputDragTarget` interface, including methods for drag start, drag continuation, and drag stop. In place of a source display, the methods use a source scene graph node to define the local source space. The corresponding event splitter is the SourceDragSplitter class.

19.2.4 SourceDragFilter class

Source drag filtering is quite similar in form and function to that of input drag filtering. The SourceDragFilter class is the host for all filtering operations, with a plug-in giving it a personality. The plug-in base class, SourceDragFilterPlugin, has the same basic internal interface and is used the same way by its host as the InputDragFilterPlugin described for basic control interpretation. It also has the same limitation of being able to handle only the simplest forms of filtering, which do not involve state or time-dependent processing.

SourceDragFilterPlugin subclasses

Although all the same filter operations could be provided as for input drag filtering, the framework provides the two most common ones. The AbsoluteSourceDragPlugin class provides absolute filtering, with a 3D point specifying the origin of the drag operation The RelativeSourceDragPlugin class performs relative origin filtering, with the 3D origin being determined by the starting point of the drag. No source drag enable filter is provided because drag enabling can be handled from the input drag space—in a POCS there is no 3D mouse sensor to directly generate source drag events.

19.2.5 SourceDragMapper class

The `SourceDragMapper` class handles the conversion from source space to actuation space. Both the source-to-target and the target-to-actuation spatial conversions are included in this mapper. A method is provided to specify whether or not drags are cumulative. Consistent with other mappers in the framework, a plug-in, the SourceDragMapperPlugin class, handles the rest of the mapping details. The internal interface between the mapper and the plug-in is similar to that in the InputDragMapper class.

By definition, source-to-target mapping is a vector mapping, meaning that the source position value is treated as a vector, with direction and magnitude being transformed instead of position. As with intuitive mapping, the distinction is subtle but important when building a source drag filter chain to feed a source mapper. The mapper's input event interface is `SourceDragTarget`, and its output event interface is `ActuationTarget`.

DirectSourceDragPlugin class

The framework provides only for direct source mapping, with the DirectSourceDragPlugin class. It includes methods for specifying source value scaling and target value offset. All mapping is direct and by dimension, similar to that for input drag mapping, but here two separate mappings are used: one for source-to-source dimension mapping, and the other for target-to-actuation dimension mapping. Mapping occurs in three steps. First the source-to-source dimension mapping is applied to the source value, then the source value is spatially transformed to the target space, and finally the target-to-actuation dimension mapping is applied to the target value.

AxisAngleSourceDragPlugin class

To handle source drag rotations, the framework provides the AxisAngleSourceDragPlugin subclass of DirectSourceDragPlugin. To accommodate the generalized mapping from source to target space, a simple fixed-axis rotation can not be used. Instead, a fixed rotation axis is defined relative to the source space, and the axis is spatially transformed as a vector to the target space. The angle of rotation defined by the source value, however, is not spatially transformed as would normally be done in the superclass. Instead, it is mapped directly, by dimension, from source to actuation spacing using the combination of the source and target mappings.

19.3 DISPLAY-RELATIVE MAPPING

DRM is a dynamic coordinate mapping technique that controls a target object according to display relative inputs, such as from the mouse or arrow keys. In terms of source and target spaces, DRM generally uses the view itself as the source space node and the actuator's target transform group is used as the target space node.

At a conceptual level, the DRM process can be summarized as follows. When the user moves the drag cursor left in the display, the intuitive drag mapper interprets that motion as occurring in 3D in the plane of the view's display. In turn, the source drag mapper obtains the view's local-to-Vworld transform, and transforms the local display movement into the equivalent motion expressed in absolute world space coordinates. The mapper then obtains the actuator target's local-to-Vworld transform, inverts it, and then transforms the world space movement into an equivalent motion in the target's local coordinate space. If, for example, the target actuator performs geometric translation, then the target object will always appear to move left in the display regardless of the position or direction of the view.

Table 19.3 summarizes the classes associated with DRM intuitive mapping. Figure 19.4 illustrates the relationship of these classes. The next example will demonstrate how to configure these and the rest of the intuitive mapping building blocks for DRM.

Table 19.3 Summary of classes related to DRM intuitive mapping

Class/Interface	Description	Input Events	Output Events
IntutiveDragMapper	A coordinate mapper for input drag to source drag conversion. Uses a plugin to perform a specific type of coordinate mapping	InputDragTarget	SourceDragTarget
IntuitiveDragMapperPlugin	The abstract base class for intuitive drag mapper plugins		
DrmPlugin	An intuitive drag mapper plugin that maps the 2D input drag to the Z=0 plane of the source space		

19.3.1 DrmDragPlugin class

For support of DRM, the framework provides the DrmDragPlugin subclass of IntuitiveDragMapperPlugin. The plug-in has very little to do. Because in DRM the intent is for the user's control inputs to be interpreted relative to the display space, the plug-in simply maps the 2D drag input to the Z=0 plane of the 3D source space. In other words, for DRM, the source space is the view space with the input drag confined to its display plane. (The intuitive drag mapper plug-in will play a more prominent role in WRM than it does in DRM.)

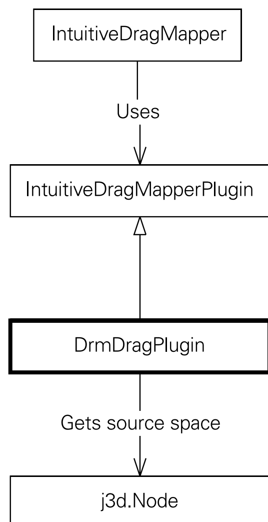

Figure 19.4 Relationship of classes related to DRM drag mapping

The constructor for DrmDragPlugin gives you a choice for how the source node is selected. Either you can let the input event's source display define it, in which case the source display's ViewingPlatform is used, or you can set it explicitly, such as with a node in the same space as the view. In any case, the source node is passed along with the mapper's output SourceDragTarget event for use by the source drag mapper.

19.3.2 Example: DrmMapping

This example demonstrates DRM of control inputs.

See

The virtual world contains four target objects as it initially displays: three in a central column (red, green, blue) and one to the right (magenta). A screen shot is provided in figure 19.5.

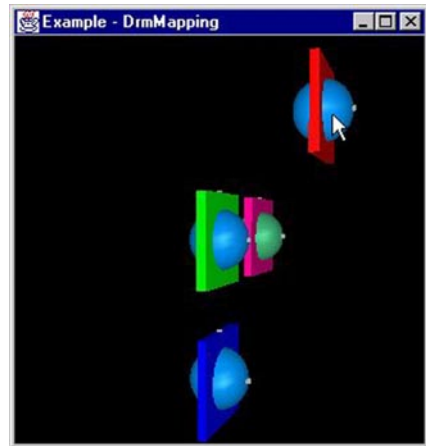

Figure 19.5 Screen shot of the DrmMapping example.

Do

- Drag the mouse (first button with SHIFT) in the display or use the ARROW keys (with SHIFT) to orbit the view in heading and elevation about the world origin.
- Drag the mouse (first button) on the top (red) target or use the ARROW keys to translate it along the display *X-Y* axes.
- Drag the mouse (first button) on the middle (green) target or use the ARROW keys to rotate it about the display *Y* axis.
- Drag the mouse (first button) on the bottom (blue) target or use the ARROW keys to scale it along the display *X-Y* axes.
- Drag the mouse (first button) on the right (magenta) target or use the ARROW keys to roll the target like a trackball relative to the display.
- Repeat the drag operations from different view directions.

Observe

- All target operations use mouseover to enable the operation.
- All coordinate mapping uses DRM, meaning that manipulation of targets occurs relative to the display and will always seem intuitive regardless of view direction.

The first code sample below is from the DrmMapping example class. It shows only those portions pertaining to the translation target and DRM mapping. DRM mapping is provided by the utility building block buildDrmTranslationMapper from the IntuitiveBlocks class, which is shown in the second code sample. This block and

others like it combine the framework's intuitive mapping blocks into individual modules specialized for a given type of geometric manipulation.

Notice in the example code how two different forms are used to specify the DRM mapper's source space reference node. The first case, for DRM translation, uses dynamic source node specification. It specifies the source node as null, indicating that the source node is to be defined by the view associated with the input drag event's source display. For this form, the local-to-Vworld read capability must be set on all the view platforms in the application from which the mapper might receive events. The second one, for DRM rotation, uses static source node specification. It sets the source node explicitly as the view object, in which case the transform read capability is automatically set on the view object by the DRM plug-in. This approach is simpler, but can only be used if the application has a single view.

DRM mapping, with dynamic and static source view

Filename: J3duiBook/examples/OverEnabling/DrmMapping.java

```
...
     // setup manipulation targets
     /// translation target
     AffineGroup affineXlt = new AffineGroup(new TestThing(
      TestThing.BOX_TRANSLATION, TestThing.BALL_DEFAULT));

     affineXlt.getTranslation().initActuation(
      new Vector4d(0, 3, 0, 0));
     getWorld().addSceneNode(affineXlt);
     ...
     // setup manipulation controls
     /// manipulation draggers, first button
     InputDragSplitter relDrag = new InputDragSplitter();

     ActuationBlocks.buildRelativeDragger(relDrag,
      getView(), Input.BUTTON_FIRST,
      Input.MODIFIER_NONE, Input.MODIFIER_NONE);

     /// manipulation mappers
     InputDragTarget mapper;

     //// Ex: let event source display define source node
     getView().getView().getViewPlatform().setCapability(
      Node.ALLOW_LOCAL_TO_VWORLD_READ);

     mapper = IntuitiveBlocks.buildDrmTranslationMapper(
      affineXlt, null, true);
     relDrag.addEventTarget(mapper);

     //// Ex: use view object as source node
     mapper = IntuitiveBlocks.buildDrmRotationYMapper(
      affineRot, getView(), true);
     relDrag.addEventTarget(mapper);
  ...
```

Utility block with mapping for DRM translation

Filename: J3duiBook/lib/j3dui/utils/blocks/IntuitiveBlocks.java

```
...
   /**
   Creates an intuitive drag mapper and corresponding source
   drag mapper, configures them for DRM translation in view X-Y,
   and connects them to the target object.
   @param target Target object.
   @param view Source display.  If null drag source is used.
   @param cumulative True if drag actions are cumulative.
   @return New building block.
   */
   public static final InputDragTarget
    buildDrmTranslationMapper(AffineGroup target,
    AppView view, boolean cumulative) {

      // use target translation node as source space
      DirectSourceDragPlugin plugin =
       new DirectSourceDragPlugin(
       target.getTranslation().getTargetNode());

      // build source mapper for target X-Y translation
      SourceDragMapper mapper = new SourceDragMapper(
       target.getTranslation(), plugin);
      mapper.setCumulative(cumulative);

      // build DRM mapper
      DrmDragPlugin drmPlugin = new DrmDragPlugin();
      if(view!=null) drmPlugin = new DrmDragPlugin(view);

      IntuitiveDragMapper drmMapper = new IntuitiveDragMapper(
       mapper, drmPlugin);

      // scale input drag
      return new InputDragFilter(drmMapper,
       new ScaleInputDragPlugin(new Vector2d(.025, .025)));
   }
...
```

19.4 WORLD-RELATIVE MAPPING

WRM is a dynamic coordinate mapping technique that controls a target object according to world-relative inputs, such as the projection of the drag cursor position onto a floor or wall in the scene. In terms of source and target spaces, the most general form of WRM uses object picking to determine the current source space node, and the actuator's target transform group as the target space node.

At a conceptual level, the WRM process can be summarized as follows: When the user moves the drag cursor over an object in the scene, the intuitive drag mapper

interprets that motion as occurring in the scene relative to the object underneath the cursor, which is the source node. In turn, the source drag mapper obtains the source node's local-to-Vworld transform, and transforms the local movement on the node's surface into the equivalent motion expressed in absolute world space coordinates, such as movement in a northerly direction. The mapper then obtains the actuator target's local-to-Vworld transform, inverts it, and then transforms the world space movement into an equivalent motion in the target's local coordinate space. If, for example, the target actuator performs geometric translation, then the target object will always appear to move north in the world regardless of the position or direction of the view.

Table 19.4 summarizes the classes associated with WRM intuitive mapping. Figure 19.6 illustrates the relationship of these classes. The next example will demonstrate how to configure these and the rest of the intuitive mapping building blocks for WRM.

Table 19.4 Summary of classes related to WRM intuitive mapping

Class/*Interface*	**Description**	**Input Events**	**Output Events**
IntutiveDragMapper	A coordinate mapper for input drag to source drag conversion. Uses a plug-in to perform a specific type of coordinate mapping	InputDragTarget	SourceDragTarget
IntuitiveDragMapperPlugin	The abstract base class for intuitive drag mapper plug-ins.		
WrmPlugin	An intuitive drag mapper plug-in that uses a pick engine to perform the most general form of WRM. It also serves as the base class for less general WRM mappers		
QuasiWrmPlugin	A WRM plug-in that performs true WRM at the start of a drag, and uses a picking plane during the drag		
PseudoWrmPlugin	A quasi-WRM plug-in that performs a mapping that can approximate WRM. It uses a pick engine to place a picking plane on the target node for use during the drag		

19.4.1 WrmDragPlugin class

For support of WRM, the framework provides the WrmDragPlugin subclass of IntuitiveDragMapperPlugin. The class has several constructors. The single-parameter constructor is for the most general form of WRM. It uses the PickEngine object supplied

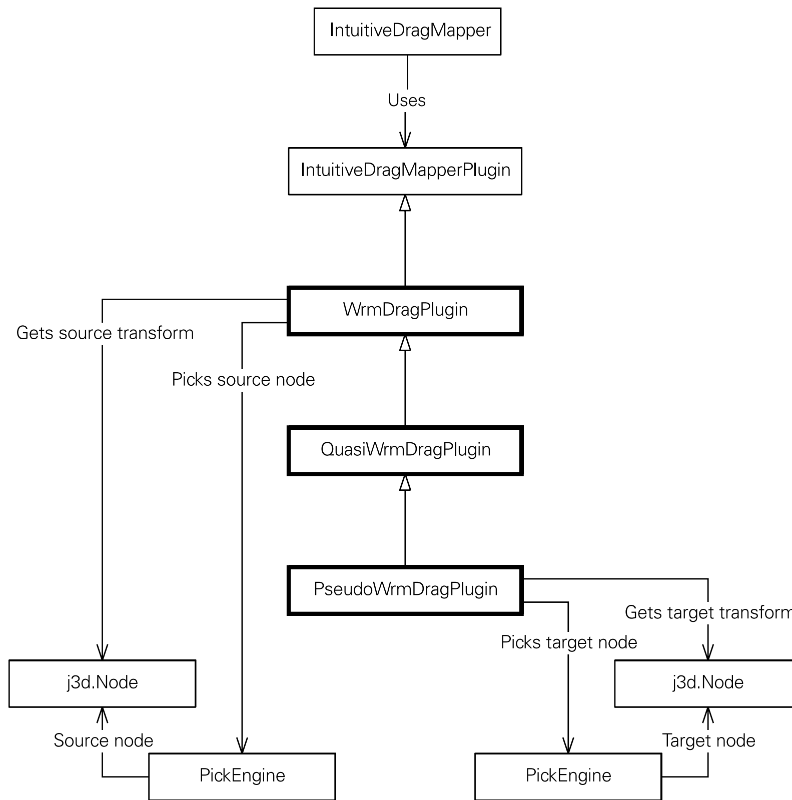

Figure 19.6 Relationship of classes related to WRM intuitive mapping

to the constructor to select the source node lying under the drag cursor, which serves as both the drag surface and the dynamic source reference space. The source space reference node is also passed along with the mapper's output SourceDragTarget event for use by the source drag mapper. The candidate source nodes with which the mapper is to work must be provided in the pick engine's target list. The mapper will generate output events only while the cursor is being dragged over a valid source target.

A second constructor, which takes a source node as its second argument, creates a WRM mapper that works much like the previous one, but the picked source nodes provide only the drag surface. The specified source node defines a static source reference space used for source mapping. A third constructor, which takes no arguments, serves as a special base class for pseudo-WRM, which is discussed in a later section.

Absolute input drag

An important note about WRM is that it will not work with a relative input drag origin. The input drag position for WRM must be absolute; otherwise, the picking that it performs will be incorrect. Because relative dragging is still a desirable feature, it

must be performed after the WRM intuitive mapping, by a source drag filter, such as a SourceDragFilter with a RelativeSourceDragPlugin.

Sticky cursor

Because the source drag position is on the surface of the source node, by default, during a translation operation the target object's origin will appear to follow the drag cursor. With this arrangement, depending on the shape of the object and the angle of view, the target object may not always appear to be stuck to the cursor—the target object may appear to drift away from the cursor as the drag progresses and the relative viewing angle changes. Sometimes this is acceptable, but often a sticky cursor will seem more intuitive than a not-so-sticky one.

Although not included in the framework, one variation on WRM that would allow it to have a sticky cursor would be to determine the hit point on the target object first, and then project a ray from the hit point to the reference source object. The position of the hit point could then be computed relative to the source reference space, and then used to make the target stick to the cursor during manipulation. Of course, if there are multiple source objects, then there is the question of which source object to use as the reference space. The answer will likely be application-specific. As you'll see shortly, the technique called pseudo-WRM can provide cursor stickiness, although the operation overall is quite restrictive in comparison to true WRM.

19.4.2 QuasiWrmDragPlugin class

True WRM, as defined by the WrmPlugin class, can be expensive because it requires continuous picking. There are also situations where a less general and more constrained form of WRM might even be desirable. The solution to both of these problems is a picking plane.

Picking plane

A picking plane offers a cheap alternative to full blown object picking if all you are interested in is a hit point. The picking plane is positioned to approximate the drag surface on a picked object. Then, during a drag, the source space hit point is computed by intersecting the same picking ray used for normal picking with the picking plane, which is significantly less demanding on the system than real picking. For situations that require it, this also constrains the source drag to a plane instead of the arbitrary surface of the source object. Instead of a picking plane, other shapes could be used that are easy to intersect with the picking ray, such as a sphere.

The framework provides utilities for handling picking planes in the Mapper class. The `buildPickPlane` utility builds a pick plane, given a hit point and a hit node defining the plane's reference space. The buildPickRay utility builds a pick ray (origin point and direction vector), given a display canvas and a pick cursor position in it. This is the same utility used by `PickEngine` for picking. The hitPickPlane utility intersects a pick ray with a pick plane and returns the 3D hit point.

Quasi-WRM

The QuasiWrmPlugin subclass of WrmPlugin offers a variation of WRM based on use of a picking plane. It uses discrete picking to select the source node at the start of a drag, the same as for general WRM, but during the drag it uses a picking plane. The discrete pick defines a source object and the initial hit point. The picking plane is placed such that it passes through the hit point and is perpendicular to the reference source space *Z* axis. Depending on the plug-in constructor used, the source reference space will be defined dynamically by the discretely picked source object, or statically by the source node specified in the constructor.

19.4.3 PseudoWrmDragPlugin class

Yet another variation on WRM is pseudo-WRM. As its name implies, it is not real WRM in that no source node picking is involved, but it does work a lot like quasi-WRM. Instead of placing a picking plane relative to the picked source node, however, it is instead placed relative to a picked target node, such as the target of the manipulation itself, the actuation target object. Because no source node picking is involved, the source space reference node must be specified statically, in the constructor. Thus, the picking plane is positioned relative to the target, through its hit point, but it is oriented according to the source node's reference space, as it is in quasi-WRM.

One of the pleasant side effects of pseudo-WRM is that the dragged target object will appear to be stuck to the drag cursor, no matter where the target is dragged or how the scene is viewed. This might seem a fortunate coincidence, but it is no coincidence. Pseudo-WRM is in essence true WRM, but with a single planar source node that always passes through the initial hit point on the target object. Thus, the projection from the hit point to the reference source object, which is what was needed for real WRM sticky cursor, is zero. With zero offset, the source point under the drag cursor is the same as the hit point on the object, and thus the target appears stuck to the cursor.

A variation of pseudo-WRM, provided by one of the constructors, is to specify the target node statically instead of using a picker. In this case, a drag operation will always use the specified target node's origin as the initial hit point, with the picking plane passing through it. In this particular form of pseudo-WRM, the target object will not always appear to be stuck to the drag cursor because of the possible offset between the cursor hit position on the object and the position of the object's origin.

19.4.4 Example: WrmMapping

This example demonstrates WRM of control inputs.

See

The virtual world contains three target objects (red, green, blue) inside a corner space formed by three orthogonal planes (a floor and two walls). A screen shot is provided in figure 19.7.

Do

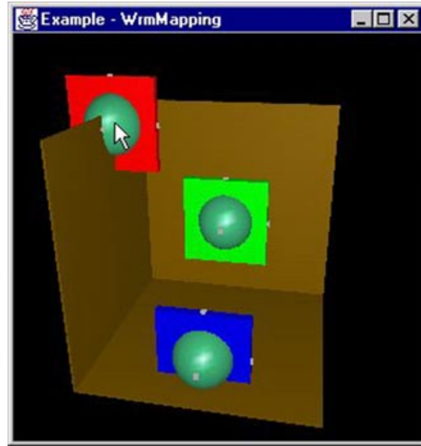

Figure 19.7 Screen shot of the WrmMapping example

- Drag the mouse (left button with SHIFT) in the display or use the ARROW keys (with SHIFT) to orbit the view in heading and elevation about the world origin.
- Drag the mouse (left button) on the left (red) target to translate it along any plane beneath the mouse cursor.
- Drag the mouse (right button) on the left (red) target to roll it relative to any plane beneath the mouse cursor.
- Drag the mouse (left button) on the middle (green) target to translate it along the plane beneath where the mouse cursor started.
- Drag the mouse (right button) on the middle (green) target to roll it relative to the plane beneath where the mouse cursor started.
- Drag the mouse (left button) on the bottom (blue) target to translate it along the bottom *X-Z* plane.
- Drag the mouse (right button) on the bottom (blue) target to roll it relative to the bottom *X-Z* plane.
- Repeat the drag operations from different view directions.

Observe

- All target operations use mouseover to enable the operation.
- The three orthogonal planes are the source objects for WRM dragging.
- The left (red) target uses full WRM coordinate mapping, meaning that target manipulation occurs relative to the source planes throughout the operation. The drag will only start and continue as long as the cursor stays over a source plane.
- The middle (green) target uses quasi-WRM coordinate mapping, meaning that target manipulation occurs relative to the source plane at the start of the operation. The drag must start over a source plane but can continue beyond it because it uses a picking plane.
- The bottom (blue) target uses pseudo-WRM coordinate mapping, meaning that target manipulation occurs relative to a predefined source object, specifically the bottom plane. The drag does not rely on the source planes to start or continue.
- For WRM and quasi-WRM translation, at drag start the target origin jumps to the source position under the cursor. The target does not appear stuck to the cursor throughout the drag.

- For pseudo-WRM translation, the target does not jump, and it appears stuck to the cursor throughout the drag regardless of viewing angle.

The first code sample following is from the WrmMapping example class. It shows only those portions that pertain to WRM manipulation. Notice that absolute input dragging is used, and that the three orthogonal planes in the scene serve as the source objects. The utility building blocks buildWrmTranslationMapper, buildWrmSphereMapper, and buildStickWrmTranslationMapper from the IntuitiveBlocks class provide WRM mapping, which are also shown below. The first two blocks take a `WrmPlugin` as one of its input parameters, and combine it with other intuitive mapping blocks, including source space relative origin filtering, to form modules specialized for a given type of geometric manipulation. The third block combines a target picker and a PseudoWrmDragPlugin in a convenient package for WRM dragging with a sticky cursor.

Notice that the WRM source picker is configured so that all hits are checked, not just the closest one. For true WRM, this allows picking during the drag to correctly determine the hit position on the underlying source planes even when the target object is dragged behind other target objects.

Setup for various forms of WRM mapping

Filename: J3duiBook/examples/OverEnabling/WrmMapping.java

```
...
    // setup WRM
    /// build source plane X
    ...
    /// build source plane Y
    ...
    /// build source plane Z
    ...
    /// build source picker
    ArrayList wrmSourceList = new ArrayList();
    wrmSourceList.add(planeX);
    wrmSourceList.add(planeY);
    wrmSourceList.add(planeZ);

    PickEngine wrmSourcePicker = new PickEngine(
     getWorld().getSceneRoot(), wrmSourceList);
    wrmSourcePicker.setHitAll(true);

    // setup manipulation controls
    /// translation draggers, first button
    InputDragSplitter absXlate = new InputDragSplitter();

    ActuationBlocks.buildAbsoluteDragger(absXlate,
     getView(), Input.BUTTON_FIRST,
     Input.MODIFIER_NONE, Input.MODIFIER_NONE);
    ...
```

```
/// manipulation mappers
InputDragTarget mapper;

//// WRM: world translation, source rotation
mapper = IntuitiveBlocks.buildWrmTranslationMapper(
 affineWrm, new WrmDragPlugin(null, wrmSourcePicker),
 false, false);
absXlate.addEventTarget(mapper);

mapper = IntuitiveBlocks.buildWrmSphereMapper(
 sphereWrm, new WrmDragPlugin(wrmSourcePicker),
 true, true);
absRotate.addEventTarget(mapper);

//// Quasi-WRM: world translation, source rotation
mapper = IntuitiveBlocks.buildWrmTranslationMapper(
 affineQWrm, new QuasiWrmDragPlugin(null,
 wrmSourcePicker), false, false);
absXlate.addEventTarget(mapper);

mapper = IntuitiveBlocks.buildWrmSphereMapper(
 sphereQWrm, new QuasiWrmDragPlugin(wrmSourcePicker),
 true, true);
absRotate.addEventTarget(mapper);

//// Pseudo-WRM: picked translation, fixed rotation
mapper = IntuitiveBlocks.buildStickyWrmTranslationMapper(
 planeY, affinePWrm, getWorld().getSceneRoot(),
 true, true);
absXlate.addEventTarget(mapper);

mapper = IntuitiveBlocks.buildWrmSphereMapper(
 spherePWrm, new PseudoWrmDragPlugin(planeY,
 affinePWrm), true, true);
absRotate.addEventTarget(mapper);
...
```

Utility blocks with mapping for common WRM operations

Filename: J3duiBook/lib/j3dui/utils/blocks/IntuitiveBlocks.java

```
...
  /**
  Creates an intuitive drag mapper and corresponding source
  drag mapper, configures them for WRM translation, and
  connects them to the target object.
  @param target Target object.
  @param wrmPlugin WRM plugin with pick engine and source
  object list.
  @param relative True if source drag actions are relative.
  Typically, use false for real WRM and true otherwise.  If
  false should initialize target geometry using
  updateActuation() instead of initActuation().
```

```
@param cumulative True if drag actions are cumulative.
Typically, use true if relative is true.
@return New building block.  Should be connected to an
absolute input drag source.
*/
public static final InputDragTarget
 buildWrmTranslationMapper(AffineGroup target,
 WrmDragPlugin wrmPlugin, boolean relative,
 boolean cumulative) {

   // build source drag mapper
   DirectSourceDragPlugin plugin =
    new DirectSourceDragPlugin(target.getTail());

   SourceDragMapper mapper = new SourceDragMapper(
    target.getTranslation(), plugin);
   mapper.setCumulative(cumulative);

   // build relative source filter
   SourceDragTarget filter;

   if(relative) {
     filter = new SourceDragFilter(
      mapper, new RelativeSourceDragPlugin());
   } else {
     filter = mapper;
   }

   // build input drag mapper
   return new IntuitiveDragMapper(filter, wrmPlugin);
}

/**
Creates an intuitive drag mapper and corresponding source
drag mapper, configures them for WRM spherical rotation, and
connects them to the target object.
@param target Target object.
@param wrmPlugin WRM plugin with pick engine and source
object list.
@param relative True if source drag actions are relative.
Typically, use false for real WRM and true otherwise.  If
false should initialize target geometry using
updateActuation() instead of initActuation().
@param cumulative True if drag actions are cumulative.
Typically, use true for rotations in general.
@return New building block.  Should be connected to an
absolute input drag source.
*/
public static final InputDragTarget
 buildWrmSphereMapper(AxisAngleSphereGroup target,
 WrmDragPlugin wrmPlugin, boolean relative,
```

```
 boolean cumulative) {

   AxisAngleSourceDragPlugin plugin;
   SourceDragMapper mapper;
   SourceDragSplitter splitter = new SourceDragSplitter();

   // X axis
   plugin = new AxisAngleSourceDragPlugin(target.getTail());
   plugin.setTargetMap(Mapper.DIM_NONE, Mapper.DIM_W,
    Mapper.DIM_NONE);
   plugin.setAxis(new Vector3d(-1, 0, 0));

   mapper = new SourceDragMapper(
    target.getAxisAngleX(), plugin);
   mapper.setCumulative(cumulative);

   splitter.addEventTarget(mapper);

   // Y axis
   plugin = new AxisAngleSourceDragPlugin(target.getTail());
   plugin.setTargetMap(Mapper.DIM_W, Mapper.DIM_NONE,
    Mapper.DIM_NONE);
   plugin.setAxis(new Vector3d(0, 1, 0));

   mapper = new SourceDragMapper(
    target.getAxisAngleY(), plugin);
   mapper.setCumulative(cumulative);

   splitter.addEventTarget(mapper);

   // build relative source filter
   SourceDragTarget filter;

   if(relative) {
     filter = new SourceDragFilter(
      splitter, new RelativeSourceDragPlugin());
   } else {
     filter = splitter;
   }

   // build input drag mapper
   return new IntuitiveDragMapper(filter, wrmPlugin);
}

/**
Creates an intuitive drag mapper and corresponding source
drag mapper, configures them for sticky pseudo-WRM
translation with a target picker, and connects them to the
target object.
@param source Reference source object defining the picking
plane orientation.
```

```
@param target Target object.
@param root Scene graph pick root containing the target.
@param relative True if source drag actions are relative.
Typically, use false for real WRM and true otherwise.  If
false should initialize target geometry using
updateActuation() instead of initActuation().
@param cumulative True if drag actions are cumulative.
Typically, use true if relative is true.
@return New building block.  Should be connected to an
absolute input drag source.
*/
public static final InputDragTarget
 buildStickyWrmTranslationMapper(Node source,
 AffineGroup target, BranchGroup root, boolean relative,
 boolean cumulative) {

  // build target picker
  ArrayList targetList = new ArrayList();
  targetList.add(target);

  PickEngine targetPicker = new PickEngine(
   root, targetList);

  // build mapper
  InputDragTarget mapper = buildWrmTranslationMapper(
   target, new PseudoWrmDragPlugin(source, targetPicker),
   relative, cumulative);

  return mapper;
}
…
```

19.5 SUMMARY

This chapter completes the presentation of the framework's implementation of 3D UI control techniques. The workhorse of a 3D UI is object picking. For the purposes of control, it forms the basis of intuitive control enabling, where user interaction is directed only to the target object to which the mouse is pointing. Earlier chapters presented an abbreviated form of the control chain that stretches from input device to target actuator. This chapter generalized that chain by introducing several additional links, with the result being a more intuitive interactive experience for the user. Two new spaces were introduced: the source drag space and the target drag space. In support of these spaces, the framework provides several new building blocks, including an intuitive drag mapper and a source drag mapper, and a new filter for operating in the 3D source drag space. Examples demonstrated the control techniques of DRM and WRM that were introduced in part 2 of the book. The examples also illustrated how these techniques can be implemented using the framework core and utility building blocks.